**Landscape**

**Test File**

My test file is generated as follows:

* Row 1: row size, column size
* Row 2: water level, w
* Row 3: gradient threshold, g
* Row 4: row and col coordinates for point A
* Row 5: row and col coordinates for point B
* Remaining numbers: represent the grid

**Global Variables**

* int rowSize, colSize: row size and column size of the grid
* int w, g: water level and gradient threshold
* Point a, b: Point a and b
* int[][] landscape: the grid that represents the landscape, each value is the z value
* int[][] visited: a helper grid that keeps track of visited and unvisited locations on the grid
* ArrayList lakes: an ArrayList of lakes to store all the lakes that are found

**Classes**

* Landscape: represents the whole landscape and the associated methods and classes with it
* Point: represents a point on the grid. Stores the row coordinates, column coordinates, the z value, and a parent pointer to use when backtracking the shortest path generated by the BFS for question 3. printPoint() prints the point.
* Lake: represents a lake. Essentially is an ArrayList of points. Stores the surface area and volume of the lake. addPoint(Point p) will add a point to the lake. computeSA() will compute the surface area of the lake. computeVol() will compute the volume of the lake. printLake() will print the Points in the lake.
* Printer: a separate helper class that contains all print functions to make the code look cleaner

**High Level Algorithm Overview**

1. Parse test file and populate global variables
2. findLakes() - Iterate through each point in the grid and check to see if it contains water and if the point is unvisited. If so, perform a BFS on the point.
3. BFS(int i, int j) - Perform a BFS on the point find all adjacent points that contain water. Store points in a Lake, store Lake in ArrayList lakes.
4. findLargestSALake() - Iterate through ArrayList lakes to find lake with largest surface
5. findLargestVolLake() - Iterate through ArrayList lakes to find lake with largest volume
6. findPath() - Clear the visited grid and set all values back to 0 (unvisited). Perform a BFS starting from Point A, but this time, keep track of each Point P that discovered the Point newP using the parent pointer. Check to see if each dequeued Point is Point B. If so, exit the loop. Retrace the path from Point B back to Point A using each Point's parent pointer and add the Points into ArrayList path. Print out ArrayList path in reverse order to get the shortest path from Point A to Point B.

**Other Notes**

* The program does not check for every single edge case so it is not robust. I am assuming that the test file will always be there and will always have the correct values and proper format. For example, checking to see if the test file exists (I throw a FileNotFoundException), checking for negative values, checking that the file only contains ints, checking to see if the grid contains a valid number of points, and so much more. I purposely did not implement checking for every single edge case just because I think it detracts from the main point of the assignment and to make the code more understandable. But I know robustness is needed in real life development.
* test.txt is just a small landscape I used for testing each step of the program design. test2.txt is a randomly generated and bigger 8x8 landscape.
* You can change the test file in line 227, and uncomment line 250 to print the grid.
* Finding all the lakes can also be done using a depth first search that will provide similar worst case time complexity.
* In findPath(), I can optimize it further by removing the O(RC) time complexity nested loop that clears int[][] visited, where R is row size and C is col size, by creating another visited 2D array. This is a trade off between O(RC) time for additional O(RC) space.